A Verification System for File Processing Programs (Mathematical Methods in Software Science and Engineering : Third Conference)

AUTHOR(S):

TAMAI, TETSUO; FUKUNAGA, KOICHI

CITATION:

TAMAI, TETSUO ...[et al]. A Verification System for File Processing Programs (Mathematical Methods in Software Science and Engineering : Third Conference). 数理解析研究所講究録 1981, 436: 198-217

ISSUE DATE:

1981-09

URL:

http://hdl.handle.net/2433/102754

数理解析研究所講究録
436巻 1981年 198-217

198

A Verification System for File Processing Programs

by

Tetsuo Tamai

Koichi Fukunaga

Mitsubishi Research Institute, Inc.

Time & Life Building
3-6, 2-chome, Ohtemachi
Chiyoda-ku, Tokyo, 100

1. Introduction

It is widely recognized even among average programmers that the verification technique is, in principle, one of the most fundamental approaches to ensure software quality. Yet, most programmers suspect its applicability to the practical situations. The points they specify as the defects of the technique are:

i) most verification methods use the notation which is not familiar to programmers (such as the predicate calculus), so that they have trouble in understanding and applying the methods;

ii) only small programs can be verified by current verifiers and considerable amount of efforts for detailed logical reasoning seems to be required to prove rather simple properties of programs. It seems impossible to manage the verification tasks for programs of practical size.

To cope with these problems, experiments of verifying non-trivial programs by practitioners are indispensable. We, members of technical staff of a software-house, have been exploring the effective way of verification for these four years. In this paper, we present the overview of our verification project. First, we examine the current status of conventional verification systems and summarize our approach in the next section. In the following sections, we show the outline of a simple verifier for file processing programs which we have developed. Some remarks on our verifier and the future research follow.

2. Approach to the practical verification system

Current verification systems

The most frequently used verification method is that of inductive assertions [3]. In this method, assertions about the relation among the program variables are placed in the text of a program such that every loop is cut by an assertion (called a loop invariant). If we can show that whenever the control passes the place of an assertion, the assertion is true with the current values of the program variables, we can claim that the program is correct with respect to the assertions.

The notion of assertion itself is intuitively understandable and this method seems to be the most natural way for programmers to state and verify their intention about their programs.

Nevertheless they have difficulty in preparing assertions, especially in finding loop invariants. This is because they don't know how to translate the description of what a program is to do into a formal specification. They need some kind of help.

There are several existing verification systems: King [9], Deutsch [2], Stanford [11], [12], USC-ISI [4], and Boyer and Moore [1]. With the notable exception of the work by Boyer and Moore, these verification systems are based on the method of inductive assertions. They slightly differ with each other in the following respects:

- methods used to generate verification conditions,
- facilities provided for user interaction.

King's pioneering system could verify only small number of programs and demonstrated the fundamental limit of fully automatic systems. Deutsch's system, named PIVOT, has some user interaction facilities and has verified essentially all of the examples in King's thesis plus several important ones. PIVOT works forward along the program paths to generate verification conditions and makes on-the-fly deduction in this generation phase. This scheme which is unique to PIVOT is

agreeable in that it simulates the ordinary execution sequence of programs and the meaning of the generated conditions is easy to understand. A defect of the forward generation is that it tends to generate irrelevant conditions for the proof, as it is not guided by the goal assertions. Later, this scheme was generalized by King [10] and named "Symbolic Execution."

Both the Stanford system and the ISI system deal with Pascal programs and use the same verification condition generator [8]. This generator is based on the axiomatic definition of Pascal [7] and works backward along the program paths. No deduction is made in this phase. Although this method is more efficient than that of PIVOT, the generated verification conditions are not easy to understand. It is difficult to debug incorrect programs by using this kind of information.

These systems have powerful facilities for simplification and theorem proving. These facilities can utilize the user's knowledge about the problem domain. Thus they succeed in verifying a variety of programs. But substantial understanding of the verification system is needed to encode the knowledge into the appropriate form required by the system.

Our approach

Here, we summarize the approach we have taken to realize a prototype verifier capable of verifying non-trivial production programs in an intuitively understandable way.

First, we tried to verify typical programs in production environment using hand simulation and to accumulate the experiences. Our aim in this stage was to analyze the primary difficulties encountered by programmers during verification and to collect the knacks of verification, if any.

One of the main observations of the above study was that the major difficulty in verifying practical programs lies within the fact that the principal programming concepts in the application area are not well

studied and their properties are not formally stated. In such cases, programmers have trouble in stating the precise specification of their programs. Thus they are far from being able to verify their programs.

In the next stage, we selected a specific domain, the sequential file processing problem, and tried to formalize the primary concepts in the domain and to verify programs using these formalized concepts [13].

An example of the formalized concepts is the order of records in a file, that is, whether they are sorted in the ascending order of the values of their key fields or not. Using these concepts, typical file processing problems, such as merging several files into one file or updating a master file by transaction records, were formally specified and the programs realizing these specifications were successfully verified.

Through these experiments of specification and verification, the usefulness of the formal approach was confirmed. The importance of the notion of abstract files, which represents intermediate files and operations on them in a conceptually integrated way, was also recognized.

Finally, we have constructed a prototype verifier for file processing problems, to demonstrate that the fairly simple tool could be useful if the problem domain is well studied and the knowledge of the domain is incorporated within the tool.

The verifier deals with programs written in a simple language called VFPL (Verifiable File Processing Language) and has the following features:

i) it uses the symbolic execution technique to reason about the properties of programs. So, programmers can follow the verification process easily;

ii) the program states are represented by the formula consisting of the predicates defined in the previous stage of this work and the execution semantics of VFPL statements are given in terms of predicate transformation rules. Programmers know the meaning of these predicates very well.

3. Overview of the VFPL verifier

Objectives

As described in the previous section, the main objective of this project was to study the feasibility of a practical verification system. What do we mean by "practical"? First of all, a practical verification system should be easy for average programmers to use. Secondly, and obviously, it must be able to show the correctness of a program that is "real" with respect to its size and its function. Moreover, a verifier should have not only a high verification capability but also a facility to support finding the cause of failure when verification does not succeed.

Features of the system

Considering the above objectives, the functions of our verifier are determined as follows.

(1) The target field of the verifier is that of sequential file processing problems.

(2) The verifier accepts programs written in VFPL, which is a special-purpose programming language to write file processing programs.

(3) The verification method is not in the line of "verification-condition-generation + theorem-proving" but is based on a kind of "symbolic execution" that transforms program states along the program execution path.

(4) The verifier is run interactively through a set of commands. Using some commands, we can interrupt symbolic execution process and inquire about the state of the program or restart symbolic execution from an arbitrary point with an arbitrary initial state. With these facilities, we can analyze program behaviors.

(5) A translator that translates a VFPL program into an equivalent COBOL program is incorporated in our verification system.

We specified the target field, because we expected to overcome the limitations faced by the current program verification technique through exploiting the knowledge of the specific problem domain. We chose the file processing problem, because it might be undesirable to choose a too narrow and uncommon area but file processing has a wide range of applications and also because it was expected that there are certain patterns in the concepts behind the file problems and the structures of their processing. The way of treating file processing problems formally is explained in detail in [13].

Configuration of the system

The overall structure of our VFPL verification system that realizes the above features is illustrated in Fig. 1. Normal use of this system goes like this. One prepares a VFPL source program and lets the *parser* parse it. The parser produces an internal code consisting of a syntax tree and a symbol table. If some extra knowledge is necessary to perform verification of this program, one represents it in the form of predicate transformation rules and supplies them to the system using the *rulehandler*. Then, one starts the *symbolic executor* to analyze the program behavior and perform verification. The symbolic executor uses the *predicate transformer* and the *simplifier* as its subsystems. When the correctness of the program is assured, one gives the VFPL program and other detail information such as data structures within records to the *translator* to generate a COBOL program. This sequence of commands are put into the system through the *command interpreter*.

Characteristics

Due to the design as described above, our VFPL verification system has such characteristics as follows.

(1) Because the verification is based on a kind of symbolic execution method, it is easy to understand the verification reasoning, following the program execution path.

(2) With the commands analogous to those found in some interactive debugger, the user can obtain much broader information about program behaviors in addition to direct results of verification.

(3) The user can write a file processing program in VFPL, a simple programming language with strong structuring facility, and verify its correctness. After the verification, he can get an executable COBOL program, supplying the VFPL program and detailed information to the translator. The verification technique has been criticized as a looking-backward technique in the sense that it tries to prove correctness of a program only after its completion. However, if used as described above, our system can be of use at the earlier stage of software development.

With these characteristics, our verification system hopefully shows a way to meet the initial objectives. The VFPL verifier currently runs under TSO on the IBM 370/168. It is coded in REDUCE, a version of LISP developed at the University of Utah by A.C.Hearn and others [6].

4. Detailed description of the system

Programming language VFPL and the parser

VFPL is a language for writing basic structures of file processing programs. It is a procedural language with file processing basic statements and minimum control structures.

Its characteristics are as follows.

(1) Files and records are defined as data types. The language is strongly typed with a facility of admitting user-defined types.

(2) VFPL has two control structures: *if-then-else* and *while-do*.

(3) There is a consturct called file-module that makes it possible to introduce virtual files. It provides a means to realize intermediate files by means of implementing *open*, *close* and *read* or *write* instructions as procedures instead of actually producing physical files. This is useful for modularizing and structuring programs.

(4) Specifications of programs can be stated in the form of input/output assertions.

The *parser* parses VFPL programs and generates syntax trees and symbol tables. As the VFPL language specification is simple, the implementation of the parser is straightforward. Some of its characteristics are strict type checking facility and some simple global analysis such as checking if an opened file is properly closed.

Symbolic execution

In this paper, we call a process of transforming the program state along the execution path, *symbolic execution*. A program state is represented by a logical formula including predicates that specify relation between files, records, and other data. For example, a formula,

program Q, then the assertion R will be true on its completion." Hoare used such axioms to specify the semantics of programming language constructs. For example, a schema of axioms

$$P^{x}_{f}\{x:=f\}P$$

defines the semantics of the assignment statement, where P is an arbitrary predicate and P^{x}_{f} is P whose all free occurrences of the variable x is replaced by the expression f. In our system, we use this kind of formulas for the purpose of axiomatically defining the meaning of predicates and functions that are introduced for specification writing and verification. For example, two axioms,

$$true\ \{open(F)\}\ osorted(state(F)),$$
$$osorted(state(F)) \wedge lastkey(state(F)) \leq R.key\ \{write(F,R)\}\ osorted(state(F)),$$

define a predicate *osorted*, which means "The output file is sorted." Here, *lastkey* is a function that returns a key of the last record written on the designated file and is assumed to have been specified beforehand in the same way. *state(F)* denotes the state of the file F.

Axioms and inference rules of a Hoare style are used for verification condition generation in many mechanical verification systems (e.g. the Stanford Pascal Verifier [11]). In verification condition generating process, the output assertion of a program is transformed following the execution path backward until it reaches the top of the path. The assertion thus obtained by the transformation must be deduced from the input assertion to show the correctness of the program and that condition is called VC (Verification Condition). In those systems, predicates and functions for the program specification and the verification, like *osorted* in the above example, are specified differently using axioms that do not include programming language constructs. They are supplied to a theorem prover that tries to prove VC's.

In our system, Hoare-like axioms are used somewhat differently. Axioms, such as those for *osorted*, play a role of program state transformation

rules. Suppose, at a point of a program just before a write statement of the form:

write(OUTF,R),

osorted(state(OUTF)) and *lastkey(state(OUTF))* $\leq$ *R.key* hold. Then, right after the execution of this *write* statement, the assertion *osorted(state(OUTF))* becomes a part of the program state. Our symbolic execution starts from the top of the program taking the given input assertion as its initial state and proceeds along the execution path, transforming the program state up to the end of the program. If the output assertion written in the program can be deduced from the final program state, then the verification of the program is successful.

In the VFPL verifier, a subsystem called the *predicate transformer* performs the function of transforming program states, applying transformation rules. During this process, another subsystem called the *simplifier* is frequently invoked and simplifies logical formulas that represent program states. This simplifier has specific knowledge of equality and partial ordering, which is of great use in reasoning about comparisons of keys that usually decide the control structure of a file processing procedure.

The simplifier not only simplifies logical formulas but also helps the symbolic executor select transformation rules applicable in the current program state or decide if an output assertion or assertions placed within the program could be deduced from the program state at that point. Details about the simplifier and also about the detection of loop invariants are given in [14].

We can summarize the features of the verification method based on this kind of symbolic execution technique as follows. Note that the program state transformation proceeds in the same direction as the program execution and knowledge on some properties of the problem, such as *osorted*, is utilized in that process. (Such knowledge is given to a theorem prover, which is independent from programs or programming languages, in conventional verifiers.) Also, the simplifier is active

during the process. Due to these characteristics, symbolic execution process gets easier to comprehend. Moreover, it can be expected that the capability of theorem proving facility need not be so sophisticated. This kind of verification process seems somewhat similar to man's way of understanding the behavior of a program and convincing oneself about its correctness.

Commands and the command interpreter

The VFPL verifier is used interactively. The user inputs commands and activates appropriate functions. These commands are interpreted by the command interpreter. Fundamental commands are those that invoke the subsystems, viz., the parser, the symbolic executor, the rulehandler, and the translator. For example, with the command

parse <file name>,

the program in the designated file will be parsed.

There is another type of commands that control the symbolic execution and display appropriate information on the process or change the execution flow. Many of these commnads have their counterparts in an interactive debugger, a software for interactively debugging a program by means of actually running it. Some typical commands are as follows:

(1) TRACE command

is a command to display the symbolic execution process with an executed source statement and the program states before and after its execution.

(2) BREAK command

is a command to set breakpoints, which are locations of a program where the symbolic execution should be interrupted and the control be transferred to the user. Besides setting breakpoints explicitly, there are implicit ways of setting them by specifying the condition to break or making the execution break at every branch or at every statement.

(3) SAVE command

is a command to save the current program state.

(4) RESTORE command

is a command to restore a saved program state and set it as the current state.

(5) EXEC command

is a command to start symbolic execution. One mode is to start the execution from the top of the program, taking the input assertion as its initial state and the other is to start from a given point with a given program state.

Translator

The *translator*, given a VFPL program and a detailed description in COBOL notation that might be hard to write in VFPL (e.g. file description in COBOL corresponding to a VFPL file variable), generates a COBOL program equivalent to the original VFPL program. The translator was developed for the purpose of making sure that our verification system as a whole could function as a useful tool for program development. Indeed, if a programmer first writes a basic structure of a file processing problem in VFPL, verifies it, adds some details to the verified program and feeds it to the translator to obtain an executable COBOL program, then the program thus developed has much more reliability.

The significance of using the programming language VFPL is that there is a way of applying symbolic execution and verification to a program written in VFPL and also that it is easy to structure a program at a high level, making use of VFPL's modularization capability. However, COBOL programmers might find the style of VFPL rather difficult to get familiar with, not to say that they feel reluctant to learn a new language. The translator can be of some help to such programmers as well, for it shows how VFPL programs are converted into COBOL programs.

Rulehandler

We explained what the program state transformation rules are and how they are applied. Some of such rules are built into the system and to enrich and systematize those rules are a necessary step toward enhancing the VFPL's capability and usefulness. However, it would still be necessary in some case for a user to prepare specific rules himself for the verification of his program. Therefore, the system is required to offer a way of registering, keeping, changing and deleting the transformation rules. It will be useful as well if it is possible to partition rules into groups and select rules by indicating group names for each verification. The rulehandler is prepared for these purposes.

For example, to enter the definition of *osorted* with a group name *OSORTED*, we first key-in the command RULE to invoke the rulehandler and then enter a subcommand of the following form:

```
CREATE OSORTED
   OS1: TRUE
        (. IO OOPEN(@F).)
        OSORTED(STATE(@F)),
   OS2: OSORTED(STATE(@F)) AND LASTKEY(STATE(@F)) <= R.KEY
        (. IO WRITE(@F,@R).)
        OSORTED(STATE(@F));
```

5. Conclusion

The VFPL Verification System currently runs on the IBM 370/168 under TSO. We already tested more than ten VFPL programs.

This system, although its target field is limited as that of file processing problems, has the following characteristics.

(1) Verification capability, powerful enough for verifying ordinary file processing programs, is realized using relatively simple mechanism.

(2) Besides verification, a means of interactively analyzing programs' behaviors is available.

(3) It is possible to generate an executable COBOL program that has been verified of its correctness with respect to its basic structure.

Some points that should be considered toward obtaining real practicality are as follows:

- to enrich our experiences of using the system and to accumulate the knowledge of the problem domain in the form of transformation rules:
- to enlarge the domain and search for the way of generalization;
- to enhance the usability of the system by way of developing an original editor within the system, optimizing the code generated by the translator, and so on;
- to study the way of introducing this system to unexperienced programmers.

Acknowledgements

This paper is based on the research entrusted to Mitsubishi Research Institute, Inc., which plays one part in "the Software Engineering Project" conducted by Joint System Development, Corp. The project is

sponsored by Japan Professional Bicycle Racing Association and supervised by Japan Software Industry Association. We wish to thank Ken Hirose, Norihisa Doi and Akinori Yonezawa for their helpful discussions and suggestions. We also thank Hajime Ishiwara, Satoshi Nishiyama and Nobuyuki Saji for their substantial work in developing the VFPL system.

References

[1] Boyer, R., and J.S.Moore, "Proving Theorems about LISP Functions," Journal of ACM, 22, 129-144 (1975).

[2] Deutsch,L.P., "An Interactive Program Verifier," Ph.D. thesis, University of California Berkley (1973).

[3] Floyd,R.W., "Assigning Meanings to Programs," Proceedings of the American Mathematical Society Symposia in Applied Mathematics, 19, 19-32, American Mathematical Society (1967).

[4] Good,D.I., R.L.London, and W.W.Bledsoe, "An Interactive Program Verification System," IEEE Transaction on Software Engineering, SE-1, 59-67 (1975).

[5] Hantler,S.L. and J.C.King, "An Introduction to Proving Correctness of Programs," Computing Surveys, 8, 331-353 (1976).

[6] Hearn,A.C., *REDUCE 2 User's Manual*, University of Utah, 1973.

[7] Hoare,C.A.R., and N.Wirth, "Axiomatic Definition of the Programming Language Pascal," Acta Informatica, 2, 335-355 (1973).

[8] Igarashi,S., R.L.London, and D.C.Luckham, "Automatic Program Verification I: a Logical Basis and its Implementation," Acta Informatica, 4, 145-182 (1975).

[9] King,J.C., "A Program Verifier," Ph.D. thesis, Carnegie-Mellon University (1969).

[10] King,J.C., "Symbolic Execution and Program Testing," Comm. ACM., 19, 385-394 (1976).

[11] Stanford Verification Group, *Stanford PASCAL Verifier User Manual*, Report No. STAN-CS-79-731, Computer Science Department, Stanford University, 1979.

[12] Suzuki,N., "Verifying Programs by Algeblaic and Logical Deduction," Proceedings of International Conference on Reliable Software, 473-481 (1975).

[13] Tamai,T. and K.Fukunaga, "Formal Treatment of File Processing Programs," Journal of Mitsubishi Research Institute, No. 8, 34-69 (1979).

[14] Tamai,T. and K.Fukunaga, "A Simplifier for Program Verification with Built-in Knowledge on Equality and Partial Ordering," Unpublished Memo (1980).